Formal Verification of Translation Validators

A Case Study on Instruction Scheduling Optimizations

Jean-Baptiste Tristan
INRIA Paris-Rocquencourt
jean-baptiste.tristan@inria.fr

Xavier Leroy
INRIA Paris-Rocquencourt
xavier.leroy@inria.fr

Abstract

Translation validation consists of transforming a program and *a posteriori* validating it in order to detect a modification of its semantics. This approach can be used in a verified compiler, provided that validation is formally proved to be correct. We present two such validators and their Coq proofs of correctness. The validators are designed for two instruction scheduling optimizations: list scheduling and trace scheduling.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs - Mechanical verification; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages - Operational semantics; D.2.4 [*Software Engineering*]: Software/Program Verification - Correctness proofs; D.3.4 [*Programming Languages*]: Processors - Optimization

General Terms Languages, Verification, Algorithms

Keywords Translation validation, scheduling optimizations, verified compilers, the Coq proof assistant

1. Introduction

Compilers, and especially optimizing compilers, are complex pieces of software that perform delicate code transformations and static analyses over the programs that they compile. Despite heavy testing, bugs in compilers (either in the algorithms used or in their concrete implementation) do happen and can cause incorrect object code to be generated from correct source programs. Such bugs are particularly difficult to track down because they are often misdiagnosed as errors in the source programs. Moreover, in the case of high-assurance software, compiler bugs can potentially invalidate the guarantees established by applying formal methods to the source code.

Translation validation, as introduced by Pnueli et al. (1998b), is a way to detect such compiler bugs at compile-time, therefore preventing incorrect code from being generated by the compiler silently. In this approach, at every run of the compiler or of one of the compiler passes, the input code and the generated code are fed to a validator (a piece of software distinct from the compiler itself), which tries to establish *a posteriori* that the generated code behaves as prescribed by the input code. The validator can use a variety of techniques to do so, ranging from dataflow analyses (Huang et al. 2006) to symbolic execution (Necula 2000; Rival 2004) to the generation of a verification condition followed by model checking or automatic theorem proving (Pnueli et al. 1998b; Zuck et al. 2003). If the validator succeeds, compilation proceeds normally. If, however, the validator detects a discrepancy, or is unable to establish the desired semantic equivalence, compilation is aborted.

Since the validator can be developed independently from the compiler, and generally uses very different algorithms than those of the compiler, translation validation significantly increases the user's confidence in the compilation process. However, as unlikely as it may sound, it is possible that a compiler bug still goes unnoticed because of a matching bug in the validator. More pragmatically, translation validators, just like type checkers and bytecode verifiers, are difficult to test: while examples of correct code that should pass abound, building a comprehensive suite of incorrect code that should be rejected is delicate (Sirer and Bershad 1999). The guarantees obtained by translation validation are therefore weaker than those obtained by formal compiler verification: the approach where program proof techniques are applied to the compiler itself in order to prove, once and for all, that the generated code is semantically equivalent to the source code. (For background on compiler verification, see the survey by Dave (2003) and the recent mechanized verifications of compilers described by Klein and Nipkow (2006), Leroy (2006), Leinenbach et al. (2005) and Strecker (2005).)

A crucial observation that drives the work presented in this paper is that translation validation can provide formal correctness guarantees as strong as those obtained by compiler verification, provided the validator itself is formally verified. In other words, it suffices to model the validator as a function $V : \mathit{Source} \times \mathit{Target} \rightarrow \mathit{boolean}$ and prove that $V(S,T) = \texttt{true}$ implies the desired semantic equivalence result between the source code S and the compiled code T. The compiler or compiler pass itself does not need to be proved correct and can use algorithms, heuristics and implementation techniques that do not easily lend themselves to program proof. We claim that for many optimization passes, the approach outlined above — translation validation *a posteriori* combined with formal verification of the validator — can be significantly less involved than formal verification of the compilation pass, yet provide the same level of assurance.

In this paper, we investigate the usability of the "verified validator" approach in the case of two optimizations that schedule instructions to improve instruction-level parallelism: list scheduling and trace scheduling. We develop simple validation algorithms for these optimizations, based on symbolic execution of the original and transformed codes at the level of basic blocks (for list scheduling) and extended basic blocks after tail duplication (for trace scheduling). We then prove the correctness of these validators

POPL'08, January 7–12, 2008, San Francisco, California, USA.

against an operational semantics. The formalizations and proofs of correctness are entirely mechanized using the Coq proof assistant (Coq development team 1989–2007; Bertot and Castéran 2004). The formally verified instruction scheduling optimizations thus obtained integrate smoothly within the Compcert verified compiler described in (Leroy 2006; Leroy et al. 2003–2007).

The remainder of this paper is organized as follows. Section 2 recalls basic notions about symbolic evaluation and its uses for translation validation. Section 3 presents the Mach intermediate language over which scheduling and validation are performed. Sections 4 and 5 present the validators for list scheduling and trace scheduling, respectively, along with their proofs of correctness. Section 6 discusses our Coq mechanization of these results. Section 7 presents some experimental data and discusses algorithmic efficiency issues. Related work is discussed in section 8, followed by concluding remarks in section 9.

2. Translation validation by symbolic execution

2.1 Translation validation and compiler verification

We model a compiler or compiler pass as a function $L_1 \rightarrow L_2 + \texttt{Error}$, where the `Error` result denotes a compile-time failure, L_1 is the source language and L_2 is the target language for this pass. (In the case of instruction scheduling, L_1 and L_2 will be the same intermediate language, Mach, described in section 3.)

Let $\leq$ be a relation between a program $c_1 \in L_1$ and a program $c_2 \in L_2$ that defines the desired semantic preservation property for the compiler pass. In this paper, we say that $c_1 \leq c_2$ if, whenever c_1 has well-defined semantics and terminates with observable result R, c_2 also has well-defined semantics, also terminates, and produces the same observable result R. We say that a compiler $C : L_1 \rightarrow L_2 + \texttt{Error}$ is *formally verified* if we have proved that

$$\forall c_1 \in L_1, c_2 \in L_2, \quad C(c_1) = c_2 \;\Rightarrow\; c_1 \leq c_2 \qquad (1)$$

In the translation validation approach, the compiler pass is complemented by a *validator*: a function $L_1 \times L_2 \rightarrow \texttt{boolean}$. A validator V is formally verified if we have proved that

$$\forall c_1 \in L_1, c_2 \in L_2, \quad V(c_1, c_2) = \texttt{true} \;\Rightarrow\; c_1 \leq c_2 \qquad (2)$$

Let C be a compiler and V a validator. The following function C_V defines a compiler from L_1 to L_2:

$$\begin{array}{lcl} C_V(c_1) & = & c_2 \text{ if } C(c_1) = c_2 \text{ and } V(c_1, c_2) = \texttt{true} \\ C_V(c_1) & = & \texttt{Error} \text{ if } C(c_1) = c_2 \text{ and } V(c_1, c_2) = \texttt{false} \\ C_V(c_1) & = & \texttt{Error} \text{ if } C(c_1) = \texttt{Error} \end{array}$$

The line of work presented in this paper follows from the trivial theorem below.

Theorem 1. *If the validator V is formally verified in the sense of (2), then the compiler C_V is formally verified in the sense of (1).*

In other terms, the verification effort for the derived compiler C_V reduces to the verification of the validator V. The original compiler C itself does not need to be verified and can be treated as a black box. This fact has several practical benefits. First, programs that we need to verify formally must be written in a programming language that is conducive to program proof. In the Compcert project, we used the functional subset of the specification language of the Coq theorem prover as our programming language. This makes it very easy to reason over programs, but severely constrains our programming style: program written in Coq must be purely functional (no imperative features) and be proved to terminate. In our verified validator approach, only the validator V is written in Coq. The compiler C can be written in any programming language, using updateable data structures and other imperative features if necessary. There is also no need to ensure that C terminates. A second benefit of translation validation is that the base compiler C can use heuristics or probabilistic algorithms that are known to generate correct code with high probability, but not always. The rare instances where C generates wrong code will be caught by the validator. Finally, the same validator V can be used for several optimizations or variants of the same optimization. The effort of formally verifying V can therefore be amortized over several optimizations.

Given two programs c_1 and c_2, it is in general undecidable whether $c_1 \leq c_2$. Therefore, the validator is in general incomplete: the reverse implication $\Leftarrow$ in definition (2) does not hold, potentially causing false alarms (a correct code transformation is rejected at validation time). However, we can take advantage of our knowledge of the class of transformations performed by the compiler pass C to develop a specially-adapted validator V that is complete for these transformations. For instance, the validator of Huang et al. (2006) is claimed to be complete for register allocation and spilling. Likewise, the validators we present in this paper are specialized to the code transformations performed by list scheduling and trace scheduling, namely reordering of instructions within a basic block or an extended basic block, respectively.

2.2 Symbolic execution

Following Necula (2000), we use *symbolic execution* as our main tool to show semantic equivalence between code fragments. Symbolic execution of a basic block represents the values of variables at the end of the block as symbolic expressions involving the values of the variables at the beginning of the block. For instance, the symbolic execution of

```
z := x + y;
t := z × y
```

is the following mapping of variables to expressions

$$\begin{array}{lcl} \texttt{z} & \mapsto & \texttt{x}^0 + \texttt{y}^0 \\ \texttt{t} & \mapsto & (\texttt{x}^0 + \texttt{y}^0) \times \texttt{y}^0 \\ v & \mapsto & v^0 \text{ for all other variables } v \end{array}$$

where v^0 symbolically denotes the initial value of variable v at the beginning of the block.

Symbolic execution extends to memory operations if we consider that they operate over an implicit argument and result, `Mem`, representing the current memory state. For instance, the symbolic execution of

```
store(x, 12);
y := load(x)
```

is

$$\begin{array}{lcl} \texttt{Mem} & \mapsto & \texttt{store}(\texttt{Mem}^0, \texttt{x}^0, 12) \\ \texttt{y} & \mapsto & \texttt{load}(\texttt{store}(\texttt{Mem}^0, \texttt{x}^0, 12), \texttt{x}^0) \\ v & \mapsto & v^0 \text{ for all other variables } v \end{array}$$

The crucial observation is that two basic blocks that have the same symbolic evaluation (identical variables are mapped to identical symbolic expressions) are semantically equivalent, in the following sense: if both blocks successfully execute from an initial state Σ, leading to final states Σ_1 and Σ_2 respectively, then $\Sigma_1 = \Sigma_2$.

Necula (2000) goes further and compares the symbolic evaluations of the two code fragments modulo equations such as computation of arithmetic operations (e.g. $1 + 2 = 3$), algebraic properties of these operations (e.g. $x + y = y + x$ or $x \times 4 = x \texttt{ << } 2$), and "good variable" properties for memory accesses (e.g.

$\texttt{load}(\texttt{store}(m, p, v), p) = v$). This is necessary to validate transformations such as constant propagation or instruction strength reduction. However, for the instruction scheduling optimizations that we consider here, equations are not necessary and it suffices to compare symbolic expressions by structure.

The semantic equivalence result that we obtain between blocks having identical symbolic evaluations is too weak for our purposes: it does not guarantee that the transformed block executes without run-time errors whenever the original block does. Consider:

```
x := 1          x := x / 0
                x := 1
```

Both blocks have the same symbolic evaluation, namely $\texttt{x} \mapsto 1$ and $v \mapsto v^0$ if $v \neq \texttt{x}$. However, the rightmost block crashes at runtime on a division by 0, and is therefore not a valid optimization of the leftmost block, which does not crash. To address this issue, we enrich symbolic evaluation as follows: in addition to computing a mapping from variables to expressions representing the final state, we also maintain a set of all arithmetic operations and memory accesses performed within the block, represented along with their arguments as expressions. Such expressions, meaning "this computation is well defined", are called *constraints* by lack of a better term. In the example above, the set of constraints is empty for the leftmost code, and equal to $\{\texttt{x}^0/0\}$ for the rightmost code.

To validate the transformation of a block b_1 into a block b_2, we now do the following: perform symbolic evaluation over b_1, obtaining a mapping m_1 and a set of constraints s_1; do the same for b_2, obtaining m_2, s_2; check that $m_2 = m_1$ and $s_2 \subseteq s_1$. This will guarantee that b_2 executes successfully whenever b_1 does, and moreover the final states will be identical.

3. The Mach intermediate language

The language that we use to implement our scheduling transformations is the Mach intermediate language outlined in (Leroy 2006). This is the lowest-level intermediate language in the Compcert compilation chain, just before generation of PowerPC assembly code. At the Mach level, registers have been allocated, stack locations reserved for spilled temporaries, and most Mach instructions correspond exactly to single PowerPC instructions. This enables the scheduler to perform precise scheduling. On the other hand, the semantics of Mach is higher-level than that of a machine language, guaranteeing in particular that terminating function calls are guaranteed to return to the instruction following the call; this keeps proofs of semantic preservation more manageable than if they were conducted over PowerPC assembly code.

3.1 Syntax

A Mach program is composed of a set of functions whose bodies are lists of instructions.

Mach instructions:

$i ::= \texttt{setstack}(r, \tau, \delta)$	register to stack move
$\mid \texttt{getstack}(\tau, \delta, r)$	stack to register move
$\mid \texttt{getparam}(\tau, \delta, r)$	caller's stack to reg. move
$\mid \texttt{op}(op, \vec{r}, r)$	arithmetic operation
$\mid \texttt{load}(chunk, mode, \vec{r}, r)$	memory load
$\mid \texttt{store}(chunk, mode, \vec{r}, r)$	memory store
$\mid \texttt{call}(r \mid id)$	function call
$\mid \texttt{label}(l)$	branch target label
$\mid \texttt{goto}(l)$	unconditional branch
$\mid \texttt{cond}(cond, \vec{r}, l_{true})$	conditional branch
$\mid \texttt{return}$	function return

Mach functions:

$f ::= \texttt{fun}\ id$
$\{\ \texttt{stack}\ n_1; \texttt{frame}\ n_2; \texttt{code}\ \vec{i}\ \}$

Mach offers an assembly-level view of control flow, using labels and branches to labels. In conditional branches `cond`, $cond$ is the condition being tested and $\vec{r}$ its arguments. Instructions are similar to those of the processor and include arithmetic and logical operations `op`, as well as memory load and stores; arguments and results of these instructions are processor registers r. *op* ranges over the set of operations of the processor and $mode$ over the set of addressing modes. In memory accesses, $chunk$ indicates the kind, size and signedness of the memory datum being accessed.

To access locations in activation records where temporaries are spilled and callee-save registers are saved, Mach offers specific `getstack` and `setstack` instructions, distinct from `load` and `store`. The location in the activation record is identified by a type τ and a byte offset δ. `getparam` reads from the activation record of the calling function, where excess parameters to the call are stored. These special instructions enable the Mach semantics to enforce useful separation properties between activation records and the rest of memory.

3.2 Dynamic semantics

The operational semantics of Mach is given in a combination of small-step and big-step styles, as three mutually inductive predicates:

Σ	$\vdash$	$\vec{i}, R, F, M \rightarrow \vec{i}', R', F', M'$	one instruction
Σ	$\vdash$	$\vec{i}, R, F, M \xrightarrow{*} \vec{i}', R', F', M'$	several instructions
G	$\vdash$	$f, P, R, M \Rightarrow R', M'$	function call

The first predicate defines one transition within the current function corresponding to the execution of the first instruction in the list $\vec{i}$. R maps registers to values, F maps activation record locations to values, and M is the current memory state. The context $\Sigma = (G, f, sp, P)$ is a quadruple of the set G of global function and variable definitions, f the function being executed, *sp* the stack pointer, and P the activation record of the caller. The following excerpts should give the flavor of the semantics.

$$\frac{v = \texttt{eval\_op}(op, R(\vec{r}))}{\Sigma \vdash \texttt{op}(op, \vec{r}, r_d) :: c, R, F, M \rightarrow c, R\{r_d \leftarrow v\}, F, M}$$

$$\frac{\begin{array}{c} true = \texttt{eval\_condition}(cond, R(\vec{r})) \\ c' = \texttt{find\_label}(l_{true}, f.code) \end{array}}{(G, f, sp, P) \vdash \texttt{cond}(cond, \vec{r}, l_{true}) :: c, R, F, M \rightarrow c', R, F, M}$$

When a `call` instruction is executed, the semantics transitions not to the first instruction in the callee, but to the instruction following the `call` in the caller. This transition takes as premise an execution $G \vdash f, P, R, M \Rightarrow R', M'$ of the body of the called function, as shown by the following rules.

$$\frac{G(R(r_f)) = f \qquad G \vdash f, F, R, M \Rightarrow R', M'}{(G, f, sp, P) \vdash \texttt{call}(r_f) :: c, R, F, M \rightarrow c, R', F, M'}$$

$$\frac{\begin{array}{c} \texttt{alloc}(M, 0, f.\texttt{stack}) = (sp, M_1) \\ \texttt{init\_frame}(f.\texttt{frame}) = F_1 \\ G, f, sp, P \vdash f.\texttt{code}, R, F_1, M_1 \xrightarrow{*} \texttt{return} :: c', R', F_2, M_2 \\ M' = \texttt{free}(M_2, sp) \end{array}}{G \vdash f, P, R, M \Rightarrow R', M'}$$

4. Validation of list scheduling

List scheduling is the simplest instruction scheduling optimization. Like all optimizations of its kind, it reorders instructions in the program to increase instruction-level parallelism, by taking advantage

of pipelining and multiple functional units. In order to preserve program semantics, reorderings of instructions must respect the following rules, where ρ is a *resource* of the processor (e.g. a register, or the memory):

- Write-After-Read: a read from ρ must not be moved after a write to ρ;
- Read-After-Write: a write to ρ must not be moved after a read from ρ;
- Write-After-Write: a write to ρ must not be moved after another write to ρ.

We do not detail the implementation of list scheduling, which can be found in compiler textbooks (Appel 1998; Muchnick 1997). One important feature of this transformation is that it is performed at the level of basic blocks: instructions are reordered within basic blocks, but never moved across branch instructions nor across labels. Therefore, the control-flow graph of the original and scheduled codes are isomorphic, and translation validation for list scheduling can be performed by comparing matching blocks in the original and scheduled codes.

In the remainder of the paper we will use the term "block" to denote the longest sequence of non-branching instructions between two branching instructions. The branching instructions in Mach are `label`, `goto`, `cond` and `call`. This is a change from the common view where a block includes its terminating branching instruction.

4.1 Symbolic expressions

As outlined in section 2.2, we will use symbolic execution to check that the scheduling of a Mach block preserves its semantics. The syntax of symbolic expressions that we use is as follows:

Resources:
$\rho ::= r \mid \texttt{Mem} \mid \texttt{Frame}$

Value expressions:

$t ::= r^0$	initial value of register r
$\mid \texttt{Getstack}(\tau, \delta, t_f)$	
$\mid \texttt{Getparam}(\tau, \delta)$	
$\mid \texttt{Op}(op, \vec{t})$	
$\mid \texttt{Load}(chunk, mode, \vec{t}, t_m)$	

Memory expressions:

$t_m ::= \texttt{Mem}^0$	initial memory store
$\mid \texttt{Store}(chunk, mode, \vec{t}, t_m, t)$	

Frame expressions:

$t_f ::= \texttt{Frame}^0$	initial frame
$\mid \texttt{Setstack}(t, \tau, \delta, t_f)$	

Symbolic code:
$m ::= \rho \mapsto (t \mid t_m \mid t_f)$

Constraints:
$s ::= \{t, t_m, t_f, \ldots\}$

The resources we track are the processor registers (tracked individually), the memory state (tracked as a whole), and the *frame* for the current function (the part of its activation record that is treated as separate from the memory by the Mach semantics). The symbolic code m obtained by symbolic evaluation is represented as a map from resources to symbolic expressions t, t_f and t_m of the appropriate kind. Additionally, as explained in section 2.2, we also collect a set s of symbolic expressions that have well-defined semantics.

We now give a denotational semantics to symbolic codes, as transformers over concrete states (R, F, M). We define inductively the following four predicates:

Σ	$\vdash$	$[\![t]\!](R, F, M) = v$	Value expressions
Σ	$\vdash$	$[\![t_f]\!](R, F, M) = F'$	Frame expressions
Σ	$\vdash$	$[\![t_m]\!](R, F, M) = M'$	Memory expressions
Σ	$\vdash$	$[\![m]\!](R, F, M) = (R', F', M')$	Symbolic codes

The definition of these predicates is straightforward. We show two selected rules.

$$\frac{v = \texttt{eval\_op}(op, \vec{v}) \qquad \Sigma \vdash [\![\vec{t}]\!](R, F, M) = \vec{v}}{\Sigma \vdash [\![\texttt{op}(op, \vec{t})]\!](R, F, M) = v}$$

$$\frac{\begin{array}{c}\forall r, \Sigma \vdash [\![m(r)]\!](R, F, M) = R'(r) \\ \Sigma \vdash [\![m(\texttt{Frame})]\!](R, F, M) = F' \\ \Sigma \vdash [\![m(\texttt{Mem})]\!](R, F, M) = M'\end{array}}{\Sigma \vdash [\![m]\!](R, F, M) = (R', F', M')}$$

For constraints, we say that a symbolic expression t viewed as the constraint "t has well-defined semantics" is satisfied in a concrete state (R, F, M), and we write $\Sigma, (R, F, M) \models t$, if there exists a value v such that $\Sigma \vdash [\![t]\!](R, F, M) = v$, and similarly for symbolic expressions t_f and t_m over frames and memory. For a set of constraints s, we write $\Sigma, (R, F, M) \models s$ if every constraint in s is satisfied in state (R, F, M).

4.2 Algorithm for symbolic evaluation

We now give the algorithm that, given a list $\vec{i}$ of non-branching Mach instructions, computes its symbolic evaluation $\alpha(\vec{i}) = (m, s)$.

We first define the symbolic evaluation $\alpha(i, (m, s))$ of one instruction i as a transformer from the pair (m, s) of symbolic code and constraint "before" the execution of i to the pair (m', s') "after" the execution of i.

$\texttt{update}(\rho, t, (m, s))$
$\quad = \quad (m\{\rho \leftarrow t\},\ s \cup \{t\})$

$\alpha(\texttt{setstack}(r, \tau, \delta), (m, s))$
$\quad = \quad \texttt{update}(\texttt{Frame}, \texttt{Setstack}(m(r), \tau, \delta, m(\texttt{Frame})), (m, s))$

$\alpha(\texttt{getstack}(\tau, \delta, r), (m, s))$
$\quad = \quad \texttt{update}(r, \texttt{Getstack}(\tau, \delta, m(\texttt{Frame})), (m, s))$

$\alpha(\texttt{getparam}(\tau, \delta, r), (m, s))$
$\quad = \quad \texttt{update}(r, \texttt{Getparam}(\tau, \delta), (m, s))$

$\alpha(\texttt{op}(op, \vec{r}, r), (m, s))$
$\quad = \quad \texttt{update}(r, \texttt{Op}(op, m(\vec{r})), (m, s))$

$\alpha(\texttt{load}(chunk, mode, \vec{r}, r), (m, s))$
$\quad = \quad \texttt{update}(r, \texttt{Load}(chunk, mode, m(\vec{r}), m(\texttt{Mem})), (m, s))$

$\alpha(\texttt{store}(chunk, mode, \vec{r}, r), (m, s))$
$\quad = \quad \texttt{update}(\texttt{Mem}, \texttt{Store}(chunk, mode, m(\vec{r}), m(\texttt{Mem}), m(r)), (m, s))$

We then define the symbolic evaluation of the block $b = i_1; \ldots; i_n$ by iterating the one-instruction symbolic evaluation function α, starting with the initial symbolic code $\varepsilon = (\rho \mapsto \rho^0)$ and the empty set of constraints.

$$\alpha(b) = \alpha(i_n, \ldots \alpha(i_2, \alpha(i_1, (\varepsilon, \emptyset))) \ldots)$$

Note that all operations performed by the block are recorded in the constraint set s. It is possible to omit operations that cannot fail at run-time (such as "load constant" operators) from s; we elected not to do so for simplicity.

The symbolic evaluation algorithm has the following two properties that are used later in the proof of correctness for the validator. First, any concrete execution of a block b satisfies its symbolic execution $\alpha(b)$, in the following sense.

Lemma 1. *Let b be a block and c an instruction list starting with a branching instruction. If $\Sigma \vdash (b;c), R, F, M \xrightarrow{*} c, R', F', M'$ and $\alpha(b) = (m, s)$, then $\Sigma \vdash [\![m]\!](R, F, M) = (R', F', M')$ and $\Sigma, (R, F, M) \models s$.*

Second, if an initial state R, F, M satisfies the constraint part of $\alpha(b)$, it is possible to execute b to completion from this initial state.

Lemma 2. *Let b be a block and c an instruction list starting with a branching instruction. Let $\alpha(b) = (m, s)$. If $\Sigma, (R, F, M) \models s$, then there exists R', F', M' such that $\Sigma \vdash (b;c), R, F, M \xrightarrow{*} c, R', F', M'$.*

4.3 Validation at the level of blocks

Based on the symbolic evaluation algorithm above, we now define a validator for transformations over blocks. This is a function V_b taking two blocks (lists of non-branching instructions) b_1, b_2 and returning `true` if b_2 is a correct scheduling of b_1.

$V_b(b_1, b_2) =$
  let $(m_1, s_1) = \alpha(b_1)$
  let $(m_2, s_2) = \alpha(b_2)$
  return $m_2 = m_1 \wedge s_2 \subseteq s_1$

The correctness of this validator follows from the properties of symbolic evaluation.

Lemma 3. *Let b_1, b_2 be two blocks and c_1, c_2 two instruction sequences starting with branching instructions. If $V_b(b_1, b_2) =$ `true` and $\Sigma \vdash (b_1;c_1), R, F, M \xrightarrow{*} c_1, R', F', M'$, then $\Sigma \vdash (b_2;c_2), R, F, M \xrightarrow{*} c_2, R', F', M'$.*

Proof. Let $(m_1, s_1) = \alpha(b_1)$ and $(m_2, s_2) = \alpha(b_2)$. By hypothesis $V_b(b_1, b_2) =$ `true`, we have $m_2 = m_1$ and $s_2 \subseteq s_1$. By Lemma 1, the hypothesis $\Sigma \vdash (b_1;c_1), R, F, M \xrightarrow{*} c_1, R', F', M'$ implies that $\Sigma, (R, F, M) \models s_1$. Since $s_2 \subseteq s_1$, it follows that $\Sigma, (R, F, M) \models s_2$. Therefore, by Lemma 2, there exists R'', F'', M'' such that $\Sigma \vdash (b_2;c_2), R, F, M \xrightarrow{*} c_2, R'', F'', M''$. Applying Lemma 1 to the evaluations of $(b_1;c_1)$ and $(b_2;c_2)$, we obtain that $\Sigma \vdash [\![m_1]\!](R, F, M) = (R', F', M')$ and $\Sigma \vdash [\![m_2]\!](R, F, M) = (R'', F'', M'')$. Since $m_2 = m_1$ and the denotation of a symbolic code is unique if it exists, it follows that $(R'', F'', M'') = (R', F', M')$. The expected result follows. □

4.4 Validation at the level of function bodies

Given two lists of instructions c_1 and c_2 corresponding to the body of a function before and after instruction scheduling, the following validator V checks that $V_b(b_1, b_2) =$ `true` for each pair of matching blocks b_1, b_2, and that matching branching instructions are equal. (We require, without significant loss of generality, that the external implementation of list scheduling preserves the order of basic blocks within the function code.)

$V(c_1, c_2) =$
  if c_1 and c_2 are empty:
    return `true`
  if c_1 and c_2 start with a branching instruction:
    decompose c_1 as $i_1 :: c_1'$
    decompose c_2 as $i_2 :: c_2'$
    return $i_1 = i_2 \wedge V(c_1', c_2')$
  if c_1 and c_2 start with a non-branching instruction:
    decompose c_1 as $b_1; c_1'$
    decompose c_2 as $b_2; c_2'$
      (where b_1, b_2 are maximal blocks)
    return $V_b(b_1, b_2) \wedge V(c_1', c_2')$
  otherwise:
    return `false`

To prove that $V(c_1, c_2) =$ `true` implies a semantic preservation result between c_1 and c_2, the natural approach is to reason by induction on an execution derivation for c_1. However, such an induction decomposes the execution of c_1 into executions of individual instructions; this is a poor match for the structure of the validation function V, which decomposes c_1 into maximal blocks joined by branching instructions. To bridge this gap, we define an alternate, block-oriented operational semantics for Mach that describes executions as sequences of sub-executions of blocks and of branching instructions. Writing Σ for global contexts and S, S' for quadruples (c, R, F, M), the block-oriented semantics refines the $\Sigma \vdash S \to S'$, $\Sigma \vdash S \xrightarrow{*} S'$ and $G \vdash f, P, R, M \Rightarrow R', M'$ predicates of the original semantics into the following 5 predicates:

$$\begin{array}{lll} \Sigma & \vdash & S \to_{nb} S' \quad \text{one non-branching instruction} \\ \Sigma & \vdash & S \xrightarrow{*}_{nb} S' \quad \text{several non-branching instructions} \\ \Sigma & \vdash & S \to_{b} S' \quad \text{one branching instruction} \\ \Sigma & \vdash & S \rightsquigarrow S' \quad \text{block-branch-block sequences} \\ G & \vdash & f, P, R, M \Rightarrow_{blocks} R', M' \end{array}$$

The fourth predicate, written $\rightsquigarrow$, represents sequences of $\xrightarrow{*}_{nb}$ transitions separated by $\to_b$ transitions:

$$\frac{\Sigma \vdash S \xrightarrow{*}_{nb} S'}{\Sigma \vdash S \rightsquigarrow S'}$$

$$\frac{\Sigma \vdash S \xrightarrow{*}_{nb} S_1 \qquad \Sigma \vdash S_1 \to_b S_2 \qquad \Sigma \vdash S_2 \rightsquigarrow S'}{\Sigma \vdash S \rightsquigarrow S'}$$

It is easy to show that the $\rightsquigarrow$ block-oriented semantics is equivalent to the original $\xrightarrow{*}$ semantics for executions of whole functions.

Lemma 4. *$G \vdash f, P, R, M \Rightarrow R', M'$ if and only if $G \vdash f, P, R, M \Rightarrow_{blocks} R', M'$.*

We are now in a position to state and prove the correctness of the validator V. Let p be a program and p' the corresponding program after list scheduling and validation: p' is identical to p except for function bodies, and $V(p(id).\texttt{code}, p'(id).\texttt{code}) =$ `true` for all function names $id \in p$.

Theorem 2. *Let G and G' be the global environments associated with p and p', respectively. If $G \vdash f, P, R, M \Rightarrow R', M'$ and $V(f.\texttt{code}, f'.\texttt{code}) =$ `true`, then $G' \vdash f', P, R, M \Rightarrow R', M'$.*

Proof. We show the analogous result using $\Rightarrow_{blocks}$ instead of $\Rightarrow$ in the premise and conclusion by induction over the evaluation derivation, using Lemma 3 to deal with execution of blocks. We conclude by Lemma 4. □

5. Validation of trace scheduling

Trace scheduling (Ellis 1986) is a generalization of list scheduling where instructions are allowed to move past a branch or before a join point, as long as this branch or joint point does not correspond to a back-edge. In this work we restrict the instructions that can be moved to non-branching instructions, thus considering a slightly weaker version of trace scheduling than the classical one.

Moving instructions to different basic blocks requires compensating code to be inserted in the control-flow graph, as depicted in figure 1. Consider an instruction i that is moved after a conditional instruction targeting a label l in case the condition is `true` (left). Then, in order to preserve the semantics, we must ensure that if the condition is true during execution the instruction i is executed. We insert a "stub", *i.e.* we hijack the control by making the conditional point to a new label l' where the instruction i is executed before going back to the label l.

Figure 1. The two extra rules of trace scheduling. On the left, an example of move after a condition. On the right, an example of move before a join point. On each example the code is shown before and after hijacking.

Dually, consider an instruction i that is moved before a label l targeted by some instruction `goto` (l) (right part of figure 1). To ensure semantics preservation, we must hijack the control of the `goto` into a new stub that contains the instruction i. This way, i is executed even if we enter the trace by following the `goto`.

In list scheduling, the extent of code modifications was limited: an instruction can only move within the basic block that contains it. The "unit" of modification was therefore the block, i.e. the longest sequences of non-branching instructions between branches. During validation, the branches can then be used as "synchronization points" at which we check that the semantics are preserved. What are the synchronization points for trace scheduling? The only instructions that limit code movement are the return instructions and the target of back-edges, i.e. in our setting, a subset of the labels. We also fix the convention that `call` instructions cannot be crossed. Those instructions are our synchronization points. In conclusion, the unit of modification for trace scheduling is the longest sequence of instructions between these synchronization points.

As in the case of list scheduling, we would like to build the validator in two steps: first, build a function that validates pairs of traces that are expected to match; then, extend it to a validator for whole function bodies. The problem is that a trace can contain branching instructions. Our previous block validator does not handle this. Moreover, we must ensure that control flows the same way in the two programs, which was obvious for the block validator since states were equivalent before branching instructions, but is no longer true for trace scheduling because of the insertion of compensating code along some control edges.

A solution to these problems is to consider another representation of the program where traces can be manipulated as easily as blocks were in the list of instructions representation. This representation is a graph of trees, each tree being a compact representation of all the traces eligible for scheduling that begin at *cut points* in the control-flow graph. The cut points of interest, in our setting, are

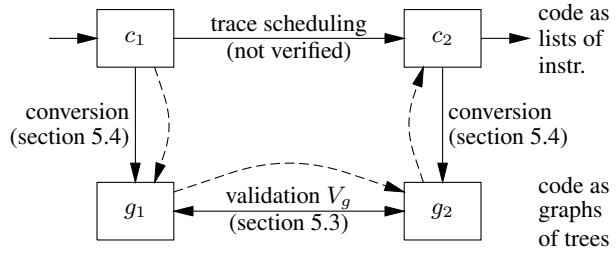

Figure 2. Overview of trace scheduling and its validation. Solid arrows represent code transformations and validations. Dashed arrows represent proofs of semantic preservation.

function entry points, calls, returns, and the labels that are targets of back-edges. The important property of these trees is that if an instruction has been moved then it must be within the boundaries of a tree.

The validator for trace scheduling is built using this program representation. To complete the validator we must transform our program given as a list of instructions into a semantically equivalent control-flow graph of trees. The transformation to this new representation also requires some code annotation. This leads to the architecture depicted in figure 2 that we will detail in the remainder of this section. Note that the transformation from lists of instructions to graphs of trees needs to be proved semantics-preserving in both directions: if the list c is transformed to graph g, it must be the case that g executes from state S to state S' if *and only if* c executes from S to S'.

5.1 A tree-based representation of control and its semantics

Figure 3 illustrates our tree-based representation of the code of a function. In this section, we formally define its syntax and semantics.

Syntax The code of a function is represented as a mapping from labels to trees. Each label corresponds to a cut point in the control-flow graph of the function. A node of a tree is labeled either by a non-branching instruction, with one child representing its unique successor; or by a conditional instruction, with two childs for its two successors. The leaves of instruction trees are $\texttt{out}(l)$ nodes, carrying the label l of the tree to which control is transferred. Finally, special one-element trees are introduced to represent call and return instructions.

Instruction trees:
$$T ::= \texttt{seq}(i, T) \qquad (i \text{ a non-branching instruction})$$
$$\mid \texttt{cond}(cond, \vec{r}, T_1, T_2)$$
$$\mid \texttt{out}(l)$$

Call trees:
$$T_c ::= \texttt{call}((r \mid id), l)$$

Return trees:
$$T_r ::= \texttt{return}$$

Control-flow graphs:
$$g ::= l \mapsto (T \mid T_c \mid T_r)$$

Functions:
$$f ::= \texttt{fun}\ id$$
$$\{\ \texttt{stack}\ n_1; \texttt{frame}\ n_2; \texttt{entry}\ l; \texttt{code}\ g;\ \}$$

Semantics The operational semantics of the tree-based representation is a combination of small-step and big-step styles. We describe executions of instruction trees using a big-step semantics

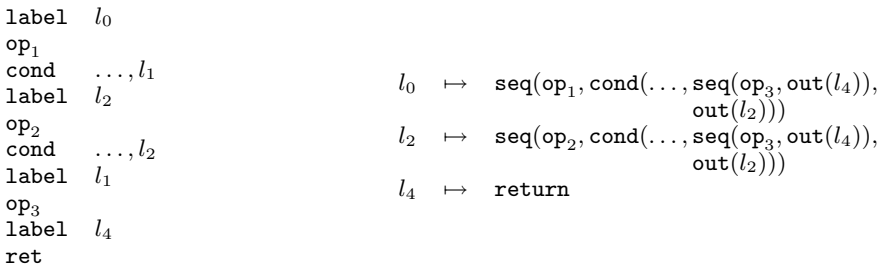

Figure 3. A code represented as a list of instructions (left) and as a graph of instruction trees (right)

$\Sigma \vdash T, R, F, M \Rightarrow l, R', F', M'$, meaning that the tree T, starting in state (R, F, M), terminates on a branch to label l in state (R', F', M'). Since the execution of a tree cannot loop infinitely, this choice of semantics is adequate, and moreover is a good match for the validation algorithm operating at the level of trees that we develop next.

$$\Sigma \vdash \mathtt{out}(l), R, F, M \Rightarrow l, R, F, M$$

$$\frac{v = \mathtt{eval\_op}(op, R(\vec{r})) \quad \Sigma \vdash T, R\{r_d \leftarrow v\}, F, M \Rightarrow l, R', F', M'}{\Sigma \vdash \mathtt{seq}(\mathtt{op}(op, \vec{r}, r), T), R, F, M \Rightarrow l, R', F', M'}$$

$$\frac{\mathtt{true} = \mathtt{eval\_condition}(cond, R(\vec{r})) \quad \Sigma \vdash T_1, R, F, M \Rightarrow l', R', F', M'}{\Sigma \vdash \mathtt{cond}(cond, \vec{r}, T_1, T_2), R, F, M \Rightarrow l', R', F', M'}$$

The predicate $\Sigma \vdash l, R, F, M \xrightarrow{*} l', R', F', M'$, defined in small-step style, expresses the chained evaluation of zero, one or several trees, starting at label l and ending at label l'.

$$\Sigma \vdash l, R, F, M \xrightarrow{*} l, R, F, M$$

$$\frac{\Sigma \vdash f.\mathtt{graph}(l), R, F, M \Rightarrow l', R', F', M' \quad \Sigma \vdash l', R', F', M' \xrightarrow{*} l'', R'', F'', M''}{\Sigma \vdash l, R, F, M \xrightarrow{*} l'', R'', F'', M''}$$

Finally, the predicate for evaluation of function calls, $G \vdash f, P, R, M \Rightarrow v, R', M'$, is re-defined in terms of trees in the obvious manner.

$$\frac{\begin{array}{c}\mathtt{alloc}(M, 0, f.\mathtt{stack}) = (sp, M_1) \\ \mathtt{init\_frame}(f.\mathtt{frame}) = F_1 \quad f.\mathtt{graph} = g \\ G, f, sp, P \vdash g(f.\mathtt{entry}), R, M_1 \xrightarrow{*} l, R', M_2 \\ g(l) = \mathtt{return} \quad M' = \mathtt{free}(M_2, sp)\end{array}}{G \vdash f, P, R, M \Rightarrow R', M'}$$

5.2 Validation at the level of trees

We first define a validator V_t that checks semantic preservation between two instruction trees T_1, T_2.

$V_t(T_1, T_2, (m_1, s_1), (m_2, s_2)) =$
  if $T_1 = \mathtt{seq}(i_1, T_1')$:
    return $V_t(T_1', T_2, \alpha(i_1, (m_1, s_1)), (m_2, s_2))$
  if $T_2 = \mathtt{seq}(i_2, T_2')$:
    return $V_t(T_1, T_2', (m_1, s_1), \alpha(i_2, (m_2, s_2)))$
  if $T_1 = \mathtt{cond}(cond_1, \vec{r}_1, T_1', T_1'')$
  and $T_2 = \mathtt{cond}(cond_2, \vec{r}_2, T_2', T_2'')$:
    return $cond_1 = cond_2 \wedge m_1(\vec{r}_1) = m_2(\vec{r}_2)$
      $\wedge\, V_t(T_1', T_2', (m_1, s_1), (m_2, s_2))$
      $\wedge\, V_t(T_1'', T_2'', (m_1, s_1), (m_2, s_2))$
  if $T_1 = \mathtt{out}(l_1)$ and $T_2 = \mathtt{out}(l_2)$:
    return $l_2 = l_1 \wedge m_2 = m_1 \wedge s_2 \subseteq s_1$
  in all other cases:
    return `false`

The validator traverses the two trees in parallel, performing symbolic evaluation of the non-branching instructions. We reuse the $\alpha(i, (m, s))$ function of section 4.2. The (m_1, s_1) and (m_2, s_2) parameters are the current states of symbolic evaluation for T_1 and T_2, respectively. We process non-branching instructions repeatedly in T_1 or T_2 until we reach either two `cond` nodes or two `out` leaves. When we reach `cond` nodes in both trees, we check that the conditions being tested and the symbolic evaluations of their arguments are identical, so that at run-time control will flow on the same side of the conditional in both codes. We then continue validation on the `true` subtrees and on the `false` subtrees. Finally, when two `out` leaves are reached, we check that they branch to the same label and that the symbolic states agree ($m_2 = m_1$ and $s_2 \subseteq s_1$), as in the case of block verification.

As expected, a successful run of V_t entails a semantic preservation result.

Lemma 5. *if* $V_t(T_1, T_2) = \mathtt{true}$ *and* $\Sigma \vdash T_1, R, F, M \Rightarrow l, R', F', M'$ *then* $\Sigma \vdash T_2, R, F, M \Rightarrow l, R', F', M'$

5.3 Validation at the level of function bodies

We now extend the tree validator V_t to a validator that operates over two control-flow graphs of trees. We simply check that identically-labeled regular trees in both graphs are equivalent according to V_t, and that call trees and return trees are identical in both graphs.

$V_g(g_1, g_2) =$
  if $\mathrm{Dom}(g_1) \neq \mathrm{Dom}(g_2)$, return `false`
  for each $l \in \mathrm{Dom}(g_1)$:
    if $g_1(l)$ and $g_2(l)$ are regular trees:
      if $V_t(g_1(l), g_2(l), (\varepsilon, \emptyset), (\varepsilon, \emptyset)) =$ `false`, return `false`
    otherwise:
      if $g_1(l) \neq g_2(l)$, return `false`
  end for each
  return `true`

This validator is correct in the following sense. Let p, p' be two programs in the tree-based representation such that p' is identical to p except for the function bodies, and $V_g(p(id).\mathtt{graph}, p'(id).\mathtt{graph}) = \mathtt{true}$ for all function names $id \in p$.

Theorem 3. *Let G and G' be the global environments associated with p and p', respectively. If* $G \vdash f, P, R, M \Rightarrow R', M'$

and $V_g(f.\texttt{graph}, f'.\texttt{graph}) = \texttt{true}$*, then* $G' \vdash f', P, R, M \Rightarrow R', M'$.

5.4 Conversion to the graph-of-trees representation

The validator developed in section 5.3 operates over functions whose code is represented as graphs of instruction trees. However, the unverified trace scheduler, as well as the surrounding compiler passes, consume and produce Mach code represented as lists of instructions. Therefore, before invoking the validator V_g, we need to convert the original and scheduled codes from the list-of-instructions representation to the graph-of-trees representation. To prove the correctness of this algorithm, we need to show that the conversion preserves semantics in both directions, or in other terms that each pair of a list of instructions and a graph of trees is semantically equivalent.

The conversion algorithm is conceptually simple, but not entirely trivial. In particular, it involves the computation of back edges in order to determine the cut points. Instead of writing the conversion algorithm in Coq and proving directly its correctness, we chose to use the translation validation approach one more time. In other terms, the conversion from lists of instructions to graphs of trees is written in unverified Caml, and complemented with a validator, written and proved in Coq, which takes a Mach function f (with its code represented as a list of instructions) and a graph of trees g and checks that $f.\texttt{code}$ and g are semantically equivalent. This check is written $f.\texttt{code} \sim g$. The full validator for trace scheduling is therefore of the following form:

$V(f_1, f_2) =$
  convert $f_1.\texttt{code}$ to a graph of trees g_1
  convert $f_2.\texttt{code}$ to a graph of trees g_2
  return $f_1.\texttt{code} \sim g_1 \wedge f_2.\texttt{code} \sim g_2 \wedge V_g(g_1, g_2)$

To check that an instruction sequence C and a graph g are equivalent, written $C \sim g$, we enumerate the cut points $l \in \mathrm{Dom}(g)$ and check that the list c of instructions starting at point l in the instruction sequence C corresponds to the tree $g(l)$. We write this check as a predicate $C, \mathcal{B} \vdash c \sim T$, where $\mathcal{B} = \mathrm{Dom}(g)$ is the set of cut points. The intuition behind this check is that every possible execution path in c should correspond to a path in T that executes the same instructions. In particular, if c starts with a non-branching instruction, we have

$$\frac{i \text{ non-branching} \qquad C, \mathcal{B} \vdash c \sim T}{C, \mathcal{B} \vdash i :: c \sim \texttt{seq}(i, T)}$$

Unconditional and conditional branches appearing in c need special handling. If the target l of the branch is a cut point ($l \in \mathcal{B}$), this branch terminates the current trace and enters a new trace; it must therefore corresponds to an $\texttt{out}(l)$ tree.

$$\frac{l \in \mathcal{B}}{C, \mathcal{B} \vdash \texttt{label}(l) :: c \sim \texttt{out}(l)}$$

$$\frac{l \in \mathcal{B}}{C, \mathcal{B} \vdash \texttt{goto}(l) :: c \sim \texttt{out}(l)}$$

$$\frac{l_{true} \in \mathcal{B} \qquad C, \mathcal{B} \vdash c \sim T}{C, \mathcal{B} \vdash \texttt{cond}(cond, \vec{r}, l_{true}) :: c \sim \texttt{cond}(cond, \vec{r}, \texttt{out}(l_{true}), T)}$$

However, if l is not a cut point ($l \notin \mathcal{B}$), the branch or label in c is not materialized in the tree T and is just skipped.

$$\frac{l \notin \mathcal{B} \qquad C, \mathcal{B} \vdash c \sim T}{C, \mathcal{B} \vdash \texttt{label}(l) :: c \sim T}$$

$$\frac{l \notin \mathcal{B} \qquad c' = \texttt{find\_label}(l, C) \qquad C, \mathcal{B} \vdash c' \sim T}{C, \mathcal{B} \vdash \texttt{goto}(l) :: c \sim T}$$

$$\frac{l_{true} \notin \mathcal{B} \qquad c' = \texttt{find\_label}(l_{true}, C) \qquad C, \mathcal{B} \vdash c \sim T \qquad C, \mathcal{B} \vdash c' \sim T'}{C, \mathcal{B} \vdash \texttt{cond}(cond, \vec{r}, l_{true}) :: c \sim \texttt{cond}(cond, \vec{r}, T', T)}$$

An interesting fact is that the predicate $C, \mathcal{B} \vdash c \sim T$ indirectly checks that $\mathcal{B}$ contains at least all the targets of back-edges in the code C. For if this were not the case, the code C would contain a loop that does not go through any cut point, and we would have to apply one of the three "skip" rules above an infinite number of times; therefore, the inductive predicate $C, \mathcal{B} \vdash c \sim T$ cannot hold. As discussed in section 6, the implementation of the $\sim$ check (shown in appendix A) uses a counter of instructions traversed to abort validation instead of diverging in the case where $\mathcal{B}$ incorrectly fails to account for all back-edges.

The equivalence check $C \sim g$ defined above enjoys the desired semantic equivalence property:

Lemma 6. *Let p be a Mach program and p' a corresponding program where function bodies are represented as graphs of trees. Assume that $p(id).\texttt{code} \sim p'(id).\texttt{code}$ for all function names $id \in p$. Let G and G' be the global environments associated with p and p', respectively. If $f.\texttt{code} \sim f'.\texttt{code}$, then $G \vdash f, P, R, M \Rightarrow R', M'$ in the original Mach semantics if and only if $G' \vdash f', P, R, M \Rightarrow R', M'$ in the tree-based semantics of section 5.1.*

The combination of Theorem 3 and Lemma 6 establishes the correctness of the validator for trace scheduling.

6. The Coq mechanization

The algorithms presented in this paper have been formalized in their entirety and proved correct using the Coq proof assistant version 8.1. The Coq mechanization is mostly straightforward. Operational semantics are expressed as inductive predicates following closely the inference rules shown in this paper. The main difficulty was to express the algorithms as computable functions within Coq. Generally speaking, there are two ways to specify an algorithm in Coq: either as inductive predicates using inference rules, or as computable functions defined by recursion and pattern-matching over tree-shaped data structures. We chose the second presentation because it enables the automatic generation of executable Caml code from the specifications; this Caml code can then be linked with the hand-written Caml implementations of the unverified transformations.

However, Coq is a logic of total functions, so the function definitions must be written in a so-called "structurally recursive" style where termination is obvious. All our validation functions are naturally structurally recursive, except validation between trees (function V_t in section 5.2) and validation of list-to-tree conversion (the function corresponding to the $\sim$ predicate in section 5.4).

For validation between trees, we used well-founded recursion, using the sum of the heights of the two trees as the decreasing, positive measure. Coq 8.1 provides good support for this style of recursive function definitions (the `Function` mechanism (Barthe et al. 2006)) and for the corresponding inductive proof principles (the `functional induction` tactic).

Validation of list-to-tree conversion could fail to terminate if the original, list-based code contains a loop that does not cross any cut point. This indicates a bug in the external converter, since normally cut points include all targets of back edges. To detect this situation and to make the recursive definition of the validator acceptable to

Coq, we add a counter as parameter to the validation function, initialized to the number of instructions in the original code and decremented every time we examine an instruction. If this counter drops to zero, validation stops on error. Appendix A shows the corresponding validation algorithm.

The Coq development accounts for approximatively 11000 lines of code. It took one person-year to design the validators, program them and prove their correctness. Figure 5 is a detailed line count showing, for each component of the validators, the size of the specifications (*i.e.* the algorithms and the semantics) and the size of the proofs.

	Specifications	Proofs	Total
Symbolic evaluation	736	1079	1815
Block validation	348	1053	1401
Block semantics	190	150	340
Block scheduling validation	264	590	854
Trace validation	234	1045	1279
Tree semantics	986	2418	3404
Trace scheduling validation	285	352	637
Label manipulation	306	458	764
Typing	114	149	263
Total	3463	7294	10757

Figure 5. Size of the development (in non-blank lines of code, without comments)

It is interesting to note that the validation of trace scheduling is no larger and no more difficult than that of list scheduling. This is largely due to the use of the graph-of-trees representation of the code. However, the part labeled "tree semantics", which includes the definition and semantics of trees plus the validation of the conversion from list-of-instructions to graph-of-trees, is the largest and most difficult part of this development.

7. Preliminary experimental evaluation and algorithmic issues

Executable validators were extracted automatically from the Coq formalization of the algorithms presented in this paper and connected to two implementations of scheduling optimizations written in Caml: one for basic-block scheduling using the standard list scheduling algorithm, the other for trace scheduling. The two verified compilation passes thus obtained were integrated in the Compcert experimental compiler (Leroy 2006; Leroy et al. 2003–2007), and tested on the test suite of this compiler (a dozen Cminor programs in the 100-1000 l.o.c. range). This test suite is too small to draw any definitive conclusion. We nonetheless include the experimental results because they point out potential algorithmic inefficiencies in our approach.

All tests were successfully scheduled and validated after scheduling. Manual inspection of the scheduled code reveals that the schedulers performed a fair number of instruction reorderings and, in the case of trace scheduling, insertion of stubs. Validation was effective from a compiler engineering viewpoint: not only manual injection of errors in the schedulers were correctly caught, but the validator also found one unintentional bug in our first implementation of trace scheduling.

To assess the compile-time overheads introduced by validation, we measured the execution times of the two scheduling transformations and of the corresponding validators. Figure 4 presents the results.

The tests were conducted on a Pentium 4 3.4 GHz Linux machine with 2 GB of RAM. Each pass was repeated as many times as needed for the measured time to be above 1 second; the times reported are averages.

On all tests except AES, the time spent in validation is comparable to that spent in the non-verified scheduling transformation. The total time (transformation + validation) of instruction scheduling is about 10% of the whole compilation time. The AES test (the optimized reference implementation of the AES encryption algorithm) demonstrates some inefficiencies in our implementation of validation, which takes about 10 times longer than the corresponding transformation, both for list scheduling and for trace scheduling.

There are two potential sources of algorithmic inefficiencies in the validation algorithms presented in this paper. The first is the comparison between the symbolic codes and constraint sets generated by symbolic execution. Viewed as a tree, the symbolic code for a block of length n can contain up to 2^n nodes (consider for instance the block $r_1 = r_0 + r_0; \ldots; r_n = r_{n-1} + r_{n-1}$). Viewed as a DAG, however, the symbolic code has size linear in the length n of the block, and can be constructed in linear time. However, the comparison function between symbolic codes that we defined in Coq compares symbolic codes as trees, ignoring sharing, and can therefore take $O(2^n)$ time. Using a hash-consed representation for symbolic expressions would lead to much better performance: construction of the symbolic code would take time $O(n \log n)$ (the $\log n$ accounts for the overhead of hash consing), comparison between symbolic codes could be done in time $O(1)$, and inclusion between sets of constraints in time $O(n \log n)$. We haven't been able to implement this solution by lack of an existing Coq library for hash-consed data structures, so we leave it for future work.

The second source of algorithmic inefficiency is specific to trace scheduling. The tree-based representation of code that we use for validation can be exponentially larger than the original code represented as a list of instructions, because of tail duplication of basic blocks. This potential explosion caused by tail duplication can be avoided by adding more cut points: not just targets of back edges, but also some other labels chosen heuristically to limit tail duplication. For instance, we can mark as cut points all the labels that do not belong to the traces that the scheduler chose to optimize. Such heuristic choices are performed entirely in unverified code (the scheduler and the converter from list- to tree-based code representations) and have no impact on the validators and on their proofs of correctness.

8. Related work

The idea of translation validation appears in the work of Pnueli et al. (1998a,b). It was initially conducted in the context of the compilation of a synchronous language. The principle of the validator is to generate verification conditions that are solved by a model checker. The authors mention the possibility of generating a proof script during model checking, which generates additional confidence in the correctness of a run of validation, but is weaker than a full formal verification of the validator as in the present paper.

The case of an optimizing compiler for a conventional, imperative language has been addressed by Zuck et al. (2001, 2003) and Barret et al. (2005). They use a generalization of the Floyd method to generate verification conditions that are sent to a theorem prover. Two validators have been produced that implement this framework: `voc-64` (Zuck et al. 2003) for the SGI `pro-64` compiler and `TVOC` (Barret et al. 2005) for the `ORC` compiler. This work addresses advanced compiler transformations, including non-structure preserving transformations such as loop optimizations (Goldberg et al. 2005) and software pipelining (Leviathan and Pnueli 2006).

Test program	List scheduling			Trace scheduling		
	Transformation	Validation	Ratio V/T	Transformation	Validation	Ratio V/T
`fib`	0.29 ms	0.47 ms	1.60	0.44 ms	0.58 ms	1.32
`integr`	0.91 ms	0.87 ms	0.96	1.0 ms	1.2 ms	1.15
`qsort`	1.3 ms	1.5 ms	1.15	1.8 ms	3.3 ms	1.89
`fft`	9.1 ms	18 ms	1.98	19 ms	62 ms	3.26
`sha1`	9.4 ms	6.7 ms	0.71	12 ms	24 ms	2.00
`aes`	56 ms	550 ms	9.76	67 ms	830 ms	12.25
`almabench`	25 ms	16 ms	0.65	56 ms	200 ms	3.57
`stopcopy`	4.1 ms	4.1 ms	1.00	4.9 ms	6.1 ms	1.25
`marksweep`	5.3 ms	6.3 ms	1.18	6.8 ms	11 ms	1.69

Figure 4. Compilation times and verification times

While the approach of Pnueli, Zuck et al. relies on verification condition generators and theorem proving, a different approach based on abstract interpretation and static analysis was initiated by Necula (2000). He developed a validator for the GCC 2.7 compiler, able to validate most of the optimisations implemented by this compiler. His approach relies on symbolic execution of RTL intermediate code and inspired the present work. Necula's validator addresses a wider range of optimizations than ours, requiring him to compare symbolic executions modulo arithmetic and memory-related equations.

Rival (2004) describes a translation validator for GCC 3.0 without optimizations. While Necula's validator handles only transformations over the RTL intermediate language, Rival's relates directly the C source code with the generated PowerPC assembly code. Rival's validator uses Symbolic Transfer Functions to represent the behaviour of code fragments. While Necula and Rival do not discuss instruction scheduling in detail, we believe that their validators can easily handle list scheduling (reordering of instructions within basic blocks), but we do not know whether they can cope with the changes in the control-flow graph introduced by trace scheduling.

Huang et al. (2006) describe a translation validator specialized to the verification of register allocation and spilling. Their algorithm relies on data-flow analyses: computation and correlation of webs of def-use sequences. By specializing the validator to the code transformations that a register allocator typically performs, they claim to obtain a validator that is complete (no false alarms), and they can also produce detailed explanations of errors.

Compared with validators based on verification condition generators, validators based on static analysis like Necula's, Rival's, Huang et al's and ours are arguably less powerful but algorithmically more efficient, making it realistic to perform validation at every compilation run. Moreover, it seems easier to characterize the classes of transformations that can be validated.

While several of the papers mentioned above come with on-paper proofs, none has been mechanically verified. There are, however, several mechanized verifications of static analyzers, i.e. tools that establish properties of one piece of compiled code instead of relating two pieces of compiled code like translation validators do, in particular the JVM bytecode verifier (Klein and Nipkow 2003) and data flow analyzers (Cachera et al. 2005).

9. Conclusions and further work

We presented what we believe is the first fully mechanized verification of translation validators. The two validators presented here were developed with list scheduling and trace scheduling in mind, but they seem applicable to a wider class of code transformations: those that reorder, factor out or duplicate instructions within basic blocks or instruction trees (respectively), without taking advantage of non-aliasing information. For instance, this includes common subexpression elimination, as well as rematerialization. We believe (without any proof) that our validators are complete, that is, raise no false alarms for this class of transformations.

It is interesting to note that the validation algorithms proceed very differently from the code transformations that they validate. The validators uses notions such as symbolic execution and block- or tree-based decompositions of program executions that have obvious semantic meanings. In contrast, the optimizations rely on notions such as RAW/WAR/WAW dependencies and back-edges whose semantic meaning is much less obvious. For this reason, we believe (without experience to substantiate this claim) that it would be significantly more difficult to prove directly the correctness of list scheduling or trace scheduling.

A direct extension of the present work is to prove semantic preservation not only for terminating executions, but also for diverging executions. The main reason why our proofs are restricted to terminating evaluations is the use of big-step operational semantics. A small-step (transition) semantics for Mach is in development, and should enable us to extend the proofs of semantic preservation to diverging executions. Another, more difficult extension is to take non-aliasing information into account in order to validate reorderings between independent loads and stores.

More generally, there are many other optimizations for which it would be interesting to formally verify the corresponding validation algorithms. Most challenging are the optimizations that move computations across loop boundaries, such as loop invariant hoisting and software pipelining.

Acknowledgments

Julien Forest helped us use the new Coq feature `Function` and improved its implementation at our request. We thank Alain Frisch for discussions and feedback.

References

Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

Clark W. Barret, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. TVOC: A translation validator for optimizing compilers. In *Computer Aided Verification, 17th Int. Conf., CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 291–295. Springer, 2005.

Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In *Functional and Logic Programming, 8th Int. Symp., FLOPS 2006*, volume 3945 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2006.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer, 2004.

David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a Data Flow Analyser in Constructive Logic. *Theoretical Computer Science*, 342(1):56–78, 2005.

Coq development team. The Coq proof assistant. Software and documentation available at `http://coq.inria.fr/`, 1989–2007.

Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.

John R. Ellis. *Bulldog: a compiler for VLSI architectures*. ACM Doctoral Dissertation Awards. The MIT Press, 1986.

Benjamin Goldberg, Lenore Zuck, and Clark Barret. Into the loops: Practical issues in translation validation for optimizing compilers. In *Proc. Workshop Compiler Optimization Meets Compiler Verification (COCV 2004)*, volume 132 of *Electronic Notes in Theoretical Computer Science*, pages 53–71. Elsevier, 2005.

Yuqiang Huang, Bruce R. Childers, and Mary Lou Soffa. Catching and identifying bugs in register allocation. In *Static Analysis, 13th Int. Symp., SAS 2006*, volume 4134 of *Lecture Notes in Computer Science*, pages 281–300. Springer, 2006.

Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, 2003.

Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *Int. Conf. on Software Engineering and Formal Methods (SEFM 2005)*, pages 2–11. IEEE Computer Society Press, 2005.

Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

Xavier Leroy et al. The Compcert certified compiler back-end. Development available at `http://gallium.inria.fr/~xleroy/compcert-backend/`, 2003–2007.

Raya Leviathan and Amir Pnueli. Validating software pipelining optimizations. In *Int. Conf. On Compilers, Architecture, And Synthesis For Embedded Systems (CASES 2002)*, pages 280–287. ACM Press, 2006.

Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

George C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation 2000*, pages 83–95. ACM Press, 2000.

Amir Pnueli, Ofer Shtrichman, and Michael Siegel. The code validation tool (CVT) – automatic verification of a compilation process. *International Journal on Software Tools for Technology Transfer*, 2:192–201, 1998a.

Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998b.

Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *31st symposium Principles of Programming Languages*, pages 1–13. ACM Press, 2004.

Emin Gün Sirer and Brian N. Bershad. Testing Java virtual machines. In *Proc. Int. Conf. on Software Testing And Review*, 1999.

Martin Strecker. Compiler verification for C0. Technical report, Université Paul Sabatier, Toulouse, April 2005.

L. Zuck, A. Pnueli, and R. Leviathan. Validation of optimizing compilers. Technical Report MCS01-12, Weizmann institute of Science, 2001.

Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A methodology for translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.

A. Appendix: the validation algorithm for conversion from instruction lists to instruction trees

We show here the algorithm that determines semantic equivalence between a list of instructions and an instruction tree, corresponding to the predicate $f, \mathcal{B} \vdash c \sim T$ in section 5.4.

```
let rec skip_control B func c counter =
  if counter = 0
  then None
  else
    match c with
    | Mlabel lbl :: c' →
        if lbl in B
        then Some (Mlabel lbl :: c')
        else skip_control B func c' ( counter − 1)
    | Mgoto lbl :: c' →
        match find_label lbl func with
        | Some c'' → if lbl in B
                     then Some (Mgoto lbl :: c')
                     else skip_control B func c'' ( counter − 1)
        | None →None
    | i :: c' → Some (i :: c')
    | _ → None

let test_out sub lbl =
  match sub with
  | out lbl ' → lbl = lbl '
  | _ → false

let rec validTreeBase B f cur t =
  let cur' = skip_control B (fn_code f) cur ( length ( fn_code f)) in
  match cur', t with
  | Some(getstack(i , t , m) :: l), getstack (i ', t ', m',sub) →
      i = i ' ∧ t = t ' ∧ m = m' ∧
      validTreeBase B f l sub
  | Some(setstack( m,i, t ) :: l), setstack (m',i ', t ', sub) →
      i = i ' ∧ t = t ' ∧ m = m' ∧
      validTreeBase B f l sub
  | Some(getparam(i,t , m) :: l), getparam(i ', t ', m',sub) →
      i = i ' ∧ t = t ' ∧ m = m' ∧
      validTreeBase B f l sub
  | Some(op(op,lr, m) :: l), op(op', lr ', m',sub) →
      op = op' ∧ lr = lr ' ∧ m = m' ∧
      validTreeBase B f l sub
  | Some(load(chk,addr, lr , m) :: l), load( chk', addr', lr ', m',sub) →
      addr = addr' ∧ chk = chk' ∧ lr = lr ' ∧ m = m' ∧
      validTreeBase B f l sub
  | Some(store(chk, addr, lr , m) :: l), store (chk', addr', lr ', m',sub) →
      addr = addr' ∧ chk = chk' ∧ lr = lr ' ∧ m = m' ∧
      validTreeBase B f l sub
  | Some(cond(c,rl, lbl ) :: l), cond(c', rl ', sub1, sub2) →
      c = c' ∧ lr = lr ' ∧
      validTreeBase B f l sub2 ∧
      (if lbl in B
       then test_out sub1 lbl
       else match find_label lbl ( fn_code f) with
            | Some l' → validTreeBase B f l' sub1
            | None → false )
  | Some(label(lbl ) :: l), out( lbl ') → lbl = lbl '
  | Some(goto(lbl) :: l), out( lbl ') → lbl = lbl '
  | _, _ → false
```